SystemVerilogで変わる論理合成ツール

――FPGAの多機能RAMやDSP機能へのマッピングを効率良く記述

デバイスの記事

Roger Do

ここではSystemVerilog言語を「論理合成ツール」の観点から解説する．SystemVerilogについては，検証記述が大幅に強化されたこともあり，検証手法に注目が集まっている．その一方で，論理合成と関係の深い設計記述についても強化されている．例えば，暗黙的なポート接続やインターフェースが新たに導入された．また，入力記述中のアサーションの内容をチェックする論理合成ツールもある．（編集部）

論理合成技術とHDL（hardware description language）は，その時代その時代の有用な技術を取り込みながら，長い年月をかけて発展してきました．その一方で，LSIは複雑化し，設計サイクルも短くなっています．デバイス技術の潜在能力を引き出すためには，新しい設計手法を積極的に活用しなければなりません．本稿では，設計生産性を向上させるため，論理合成ツールが，単純なブール代数のマッピング・ツールから，SystemVerilogのような高度な言語を取り扱うツールへと発展してきた経緯について説明します．

● 単なる論理のマッピングから複雑な最適化ツールへ発展

論理合成技術は，単純な論理ゲートのマッピングから始まりました．今日では，メモリ構造や複雑な算術演算機能，DSP（積和演算）機能，複雑なクロック構造などを取り扱うことができます．

論理合成技術は，当初，回路図入力の代わりに使用されていました，このころは，単純なブール代数式をマッピングする用途で使われていました．当時の合成対象であるSimple PLD（programmable logic device）や積項（product term）ベースのプログラマブル・デバイスは，ブール論理を実現する機能しか備えていませんでした．

大規模なFPGAが登場すると，ブール論理のマッピングだけでなく，別の最適化機能も要求されるようになりました．論理をLUT（look-up table）にマッピングすると同時に，遅延時間と面積（ゲート数）を最小化しなければなりません．また，算術演算については，キャリ・チェーン（けた上げ信号）を考慮しながら適切に実装する必要がありました．

そして，さらに大規模なFPGA（例えば，米国Xilinx社が提唱する「プラットフォームFPGA」など）を取り扱うため，より多くの複雑な論理を扱う能力と，さらに高度な最適化の機能が求められるようになりました．例えば，多機能の専用RAMブロックや複雑なDSPブロックにマッピングする機能が開発されました（**図1**）．

● 記述量が減り，モデル化・カプセル化しやすい

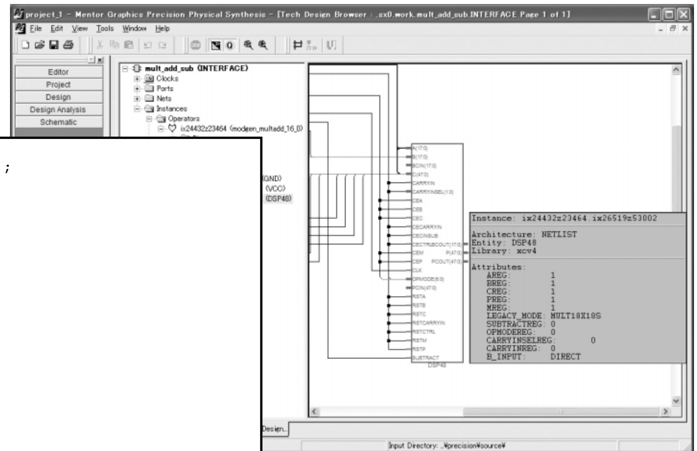
LSIはより大きく，そして従来よりもはるかに複雑になっています．設計サイクルは年々短くなり，市場への製品の早期投入（time-to-market）に対する圧力は増大しています．そこで現在，SystemVerilogに注目が集まっています．

SystemVerilogはまったく新規の言語ではなく，従来のVerilog HDLの発展形です．すなわちSystemVerilogは，Verilog 95（IEEE 1364-1995）やVerilog 2001（IEEE 1364-2001）と互換性があります．

論理合成の観点から見たSystemVerilogの利点を，以下にまとめます．

- 効率的なコーディング手法によって，RTL（register

KeyWord 論理合成，プラットフォームFPGA，SystemVerilog，IEEE 1364，Verilog HDL，VHDL，ポート接続，インスタンシエーション，インターフェース，Precision Synthesis，アサーション，ステート・マシン



```
module mult_add_sub (clk, load, dataa, datab, datac, preg);

  input  signed[7:0]  dataa;
  input  signed[7:0]  datab;
  input  signed[15:0] datac;
  input  clk, load;

  output signed[15:0] preg;
  reg    signed[15:0] mult0_result;
  reg    signed[7:0]  reg_dataa;
  reg    signed[7:0]  reg_datab;
  reg    signed[15:0] reg_datac;
  reg    signed[15:0] preg;
  wire   signed[15:0] result;

  always @(posedge clk)
  begin
    reg_datac    <= datac;
    reg_dataa    <= dataa;
    reg_datab    <= datab;
    mult0_result <= reg_dataa * reg_datab;
    preg         <= result;
  end
  assign result = load ?  reg_datac - mult0_result : reg_datac + mult0_result;

endmodule
```

図1　DSP（積和演算）機能の合成

最近の論理合成ツールは，入力した論理を，FPGAが内蔵する多機能の専用RAMブロックや複雑なDSPブロックにマッピングする機能を備えている．ここでは積和演算のモジュールをDSPブロックにマッピングした例を示している

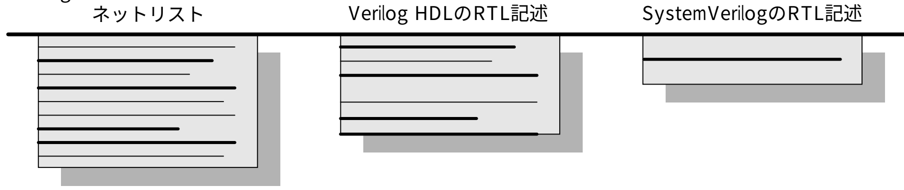


図2
SystemVerilogを使うとVerilog HDLよりも簡素な表現が可能

一般にSystemVerilogを利用すると，Verilog HDL（RTL記述の場合）の1/2～1/5の行数で同じ設計を表現できる

transfer level）記述をコンパクトに書ける．これにより設計生産性が向上する

- インターフェース記述を使って多様な抽象度のモデルを表現したり，カプセル化したりできる
- 合成とシミュレーションの言語を共通化できる
- 合成可能な任意の規模のモデルを正確に記述できる

上述のSystemVerilogの利点について，それぞれ細かく説明していきます．

● コード量がVerilog HDLの1/2～1/5に

回路規模の増大とともに，設計を表現するのに必要なRTLコードの行数が増加しています．実際に記述されるコードの行数が増えると，設計に含まれるバグも増えます．

SystemVerilogでは設計記述の構文が見直されており，より簡潔に表現できるようになりました（**図2**）．また，検証記述と設計記述のセマンティクス（意味論）の統一も図られています．これにより，設計のバグを早期に検出し，製品の市場投入時期を早められます．一般にSystemVerilogを利用すると，Verilog HDL（RTL記述の場合）の1/2～1/5の行数で同じ設計を表現できます．

●「暗黙的なポート接続」を利用できる

VHDLやVerilog HDLのポート接続には，「順序によるポート接続」と「名まえによるポート接続」があります．順序によるポート接続は，記述の順番をまちがえる危険性があります．名まえによるポート接続は，とくにトップ・レベルのモジュールで冗長かつ重複する記述となりがちです．これに対してSystemVerilogは，ポート接続の記述に`.name`や`.*`を利用できます（**リスト1**）．これらを利用するとコードが簡潔になり，ポート接続を記述する際に発生しがちな記述ミスを減らせます．

ポート接続にはタイプ（type）の問題があります．接続す

るポートのサイズは一致していなければならず，ポートの記述を省略することはできません．そして，すべてのポートが宣言されていなければなりません．これに対して，SystemVerilogの「暗黙的なポート接続(.*)」は非常に重要な働きをします．これはVHDLやVerilog HDLにはない機能です．こうした機能は，とくにASICのプロトタイプとしてFPGAを利用する際に役立ちます．設計者は，トップ・レベル(最上位階層)のインスタンシエーションのコードを最大75%も削減できます(**表1**)．また，VHDLと同じように記述ミスが生じにくいコーディング・スタイルを提供します．

暗黙的なポート接続の効果を，以下にまとめます．

リスト1 SystemVerilogのポート接続

```
module alu (
  output reg [7:0] alu_out,
  output reg zero,
  input [7:0] ain, bin,
  input [2:0] opcode);

  // RTL code for the alu module

endmodule

module alu_accum3 (
  output [15:0] dataout,
  input [7:0] ain, bin,
  input [2:0] opcode,
  input clk, rst_n);

  wire [7:0] alu_out;

  alu alu (.alu_out, .zero(), .ain, .bin, .opcode);

endmodule
```

(**a**) .nameによる接続例

```
module alu_accum4 (
  output [15:0] dataout,
  input [7:0] ain, bin,
  input [2:0] opcode,
  input clk, rst_n);

  wire [7:0] alu_out;

  alu alu (.*, .zero());

endmodule
```

(**b**) .*による接続例

リスト2 インターフェース記述

```
interface address_bus;
   parameter p1 = 7;
   wire [p1:0] read_address;
   wire [p1:0] write_address;
   modport address_port_in (input read_address,
                            input write_address);
endinterface
//-----------------------------------------------------
interface data_bus;
   parameter p2 = 7;
   wire [p2:0] data_in;
   wire [p2:0] data_out;
   modport data_port (input data_in, output data_out);
endinterface
//-----------------------------------------------------
interface control_bus;
   wire reset;
   wire clk;
   wire wren;
   wire ren;
   modport memory_control (input reset, input clk,
                           input wren, input ren);
endinterface

module memory (address_bus.address_port_in addr,
               control_bus.memory_control ctrl,
               data_bus.data_port data);
parameter p1 =7;
parameter p2 =7;

typedef logic [7:0] BYTE;
BYTE memory[(2*p1+1):0][(2*p1+1):0];
BYTE latch;

assign  data.data_out = latch;
…
…
endmodule
```

```
module top (clk, reset, wren, ren, raddr, waddr, wdata,
                                                  rdata);
parameter p1 =7;
parameter p2 =7;

input clk;
input reset;
input wren;
input ren;
input  [p1:0] raddr;
input  [p1:0] waddr;
input  [p2:0] wdata;
output [p2:0] rdata;
logic  [p2:0] rdata;

// Instantiate interfaces
address_bus #(.p1(p1)) addr();
data_bus    #(.p2(p2)) data();
control_bus            ctrl();

// Instantiate memory module
memory #(.p1(p1), .p2(p2)) mem_i (.*);

assign ctrl.clk   = clk;
assign ctrl.reset = reset;
assign ctrl.wren  = wren;
assign ctrl.ren   = ren;
assign addr.read_address  = raddr;
assign addr.write_address = waddr;
assign data.data_in       = wdata;

always_comb
begin
   rdata = data.data_out;
end
endmodule
```

- ポート接続をモデル化するのに必要なコード量を大幅に減らせる
- ポート接続において，VHDLと同様の強力なタイプを提供する
- すべてのポート・サイズを一致させなければならない．これにより，インスタンシエーションにおけるポート・サイズの記述ミスを減らせる
- 接続しないポートもリストアップしなくてはならないため，そのようなポートの情報を表現できる
- VHDLやVerilog HDLによるトップ・レベルの設計で生じがちだった記述の繰り返しが不要になる．これにより，記述に要する時間や記述ミスを減らせる

● インターフェースでモジュール間通信の詳細を隠ぺい

Verilog HDLでは，モジュール・ポートを介してモジュールどうしを接続します．この場合，設計を構成するモジュールごとにポートを定義する必要があり，実装するハードウェアに対する詳細な知識が必要になります．モジュールの中には同じようなポートを多数持つものがあり，各モジュールに対して冗長なポート定義を行わなければならない場合もあります．例えば，PCIバスに接続されるすべてのモジュールに対して，同じポート定義を行わなければなりません．

これに対して，SystemVerilogのインターフェース記述は，抽象度の高いモジュールどうしを接続する機能を提供します．インターフェースは，キーワード`interface`と`endinterface`の間に記述され，モジュールとは独立に定義されます．モジュールは，インターフェースをまさに一つのポートのように使用できます．**リスト2**に，インターフェースの定義やインスタンシエーション，接続の基礎的な構文を示します．

単純に考えると，インターフェースはワイヤ(`wire`)を一つにまとめたものと考えることもできます．しかし，インターフェースの機能は，単に接続のための信号をまとめることだけではありません．その中に，モジュールに共通する機能を含めることができます．さらに，プロトコル・チェックの機能を含めることもできます．

インターフェースは，(各モジュールに分散していた)記述を1ヵ所にまとめ，モジュールどうしの「通信」をモデル化することを目的としています．インターフェースの記述を1ヵ所にまとめ，かつ論理合成の過程で適切なハードウェアに展開できるようにすることは，設計工程全体から見てとても重要です(**図3**)．例えば，こんなことがよくあります．設計チームがバスの設計仕様を作成するのですが，統合テストの段階でその仕様にあいまいさがあることに気がつきます．設計仕様に対して2種類以上の実装方法が存在することが明らかになり，設計の一部を修正することになるのです．

表1　トップ・レベル記述の比較

	ワード数	キャラクタ数	行　数
Verilog HDLによるポート・インターフェース	250	1770	122
SystemVerilogによる暗黙的なポート・インターフェース	72	492	37

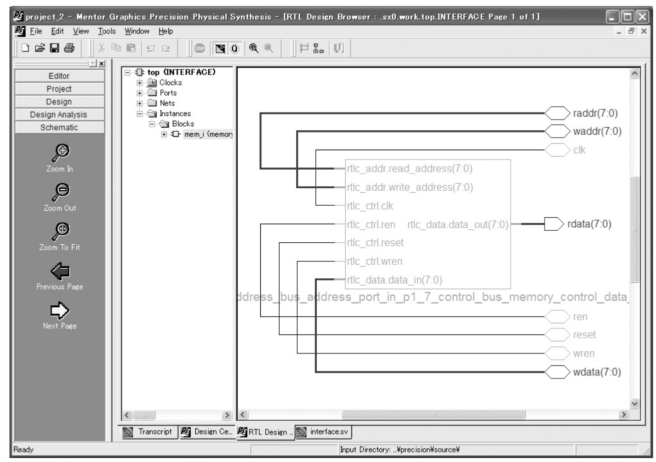


図3　インターフェース合成の結果
インターフェースは，(各モジュールに分散していた)記述を1ヵ所にまとめ，モジュールどうしの「通信」をモデル化する．ここではインターフェース記述からバスが合成された例を示している

インターフェースを使用すると，「ひとりの設計者がインターフェースの作成を担当し，(モジュール設計を担当する)ほかの設計者に対してバス接続に使用するAPI(application programming interface)を提供する」といった体制をとることができます．これにより，バス上のデータ転送の詳細を(モジュールの設計者に対して)隠ぺいできます．また，記述を1ヵ所にまとめることによる別の利点もあります．インターフェースにほかの信号を追加しなければならないときに，それに合わせてすべてのモジュールの信号を修正しなくても済むのです．

インターフェースのさらなる利点は，同じAPIをサポートする別のインターフェースと容易に交換できるということです．例えば，元の設計ではシリアル・バスで開発され

ており，パラレル・バスが必要であることがわかった場合，設計のほかの部分はそのままにして，インターフェースだけを交換することができます．これは短時間で設計をリターゲットする手法（別の構造の回路やデバイス・アーキテクチャに変換する手法）になります．

さらに，多くのモジュールが同じインターフェースを参照する場合，コード量を大幅に削減できます．各モジュール上のポートをすべてリストアップする必要はなくなり，インターフェースの中に1回記述するだけで済みます．

● 合成時にアサーションをチェック

SystemVerilogは設計のふるまいを検証するための専用記述を提供します．これにより，一つの言語で，合成とシミュレーションの両方に対応できます．

アサーションは，特定の状態やシーケンスがその設計において「つねに真である」ことを表明するステートメントです．状態やシーケンスが真でない場合は，アサーション記述をもとにエラー・メッセージが出力されます．

アサーションは設計に対する「事実の記述」であり，機能カバレッジの測定や検証用の入力スティミュラス（シミュレーション・パターン）の生成に利用されます．アサーションには，コンカレント・アサーションとイミディエート・アサーションの2種類があります．コンカレント・アサーションはクロック・サイクルをベースに，値と変数をサンプリングします．一方，イミディエート・アサーションは手続き文のように実行されます．

SystemVerilogでは，同一のコードを合成とシミュレーションの両方に使用できます．例えば米国Mentor Graphics社の論理合成ツール「Precision Synthesis」の場合，合成用のアサーションをサポートしており，実行時にアサーションの条件をチェックします．

● SystemVerilogで配列を効率良く記述

以下では，FPGAをターゲットとした場合の論理合成について説明します．

SystemVerilogは，FPGAの機能を記述する際にも非常に有効に利用できます．FPGA技術は年々複雑になっており，最近では利用できる機能を正確に記述することが重要になっています．例えば，最新のRAM機能が大規模なFPGAに使われていますが，Verilog HDLやVHDLではSystemVerilogほど効率良くこれらを記述できません．

例えば，Verilog HDLのデータ・タイプによって配列を宣言できます．タイプregとタイプnetはベクタ宣言を定義できますし，配列は何次元でも使用できます．しかし，Verilog HDLでは，1度に一つの配列要素へのアクセスしか認められていません．

以下では，この点でSystemVerilogがどのように効率的であるかを示します．この例では，データをほかの配列にコピーしています．

まず，Verilog HDLではforループを1回に1バイトしかコピーできません．

```
bit [7:0] A [1:256];
        //256 by 8-bit array declaration
bit [7:0] B [1:256];
for (i=1; i<=256; i=i+1);
    B[i] = A[i];
              copy one element at a time
```

一方，SystemVerilogではforループを使用せずに全配列をコピーできます．

```
B = A; // copy the entire array
```

SystemVerilogは，1次元配列の1個以上の連続したビット選択を参照するために部分選択を使用します．これは，Verilog HDLの部分選択の使用方法と同じです．

```
reg [63:0] data;
reg [7:0] byte2;
byte2 = data[23:16];
       // an 8-bit part select from data
```

SystremVerilogは，パックされていない配列のすべての要素に対して，1回の代入でデフォルト値の初期化が可能です．

```
r1 = {default:8'hFF};
                    // initialize an array
```

SystemVerilogは，変数のメモリ・ワードに簡単にアクセスできるように，どのような多次元配列（メモリ・ワードの表現）も使用できます．

```
logic [1:0][7:0] mem [255 : 0 ] ;
                // 256 words of 16 bit each
```

```
mem [0][1] = 8'b00110011;
                //upper byte of 0th word
mem [0][0] = 8'b00111111;
                //lower byte of 0th word
```

SystemVerilogでは，配列の連続した要素の選択にスライスを使用できます．一方Verilog HDLでは，配列の1要素しか選べず，スライスを使用することはできません．

1次元配列はVerilog 95でもサポートされていますが，HDL設計者は長年，配列構造の非効率なコーディング方法に悩まされていました．ここでは**リスト3**の配列アクセスを考えます．この中の以下の宣言に対するもっとも大きなアクセスは，SystemVerilogでは64ビット，Verilog HDLでは8ビットになります（**図4**）．

```
logic [3:0] [1:0] [7:0] DATA [3:0]
```

表2にSystemVerilogとVerilog HDLの配列の比較をまとめます．

リスト3　配列アクセスの記述例

```
module multi_array (input logic [63:0] datain,
output logic [3:0] [1:0] [7:0] dataout [3:0],
input clock, reset);

logic [3:0] [1:0] [7:0] DATA [3:0];
// Only 64 out of 256 bits of this array are used
always @ (posedge clock) begin
  if (reset) DATA <= {default:
     64'hFFFF_FFFF_FFFF_FFFF}; // initialize an array
  else begin
    DATA [3] [3] [1] <= datain[63:56];
    DATA [3] [3] [0] <= datain[55:48];
    DATA [3] [2] [1] <= datain[47:40];
    DATA [3] [2] [0] <= datain[39:32];
    DATA [3] [1] [1] <= datain[31:24];
    DATA [3] [1] [0] <= datain[23:16];
    DATA [3] [0] [1] <= datain[15:8];
    DATA [3] [0] [0] <= datain[7:0];
  end
end

assign dataout [3] [3:0] = DATA [3] [3:0];
                                // Output 64-bit data
endmodule
```

● **ステート･マシンのSystemVerilog記述を合成**

最後に，FPGAをターゲットにステート・マシン（FSM：finite state machine）を合成する場合の記述について説明します．**リスト4**にステート･マシンの記述を示します．**図5**

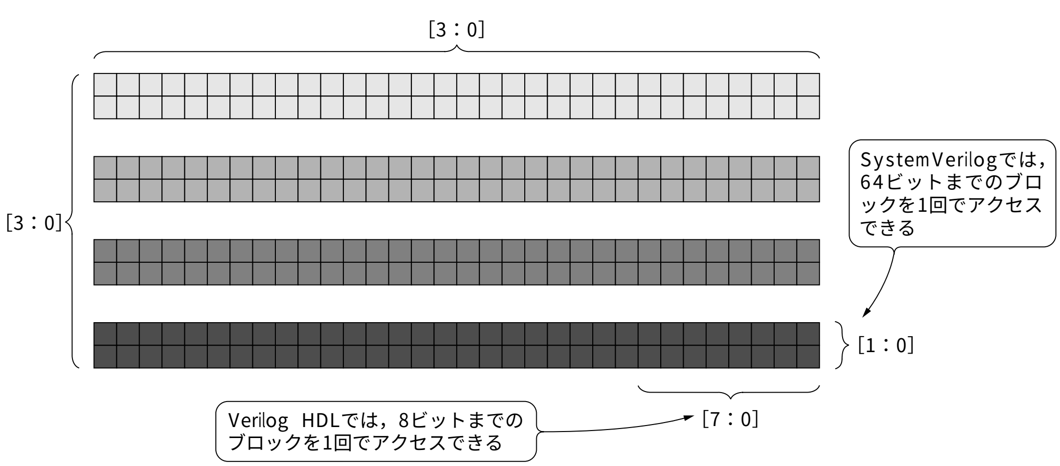


図4
SystemVerilogによる配列のアクセス
リスト3の配列アクセスの中の `logic [3:0] [1:0] [7:0] DATA [3:0]` の宣言に対するもっとも大きなアクセスは，SystemVerilogでは64ビット，Verilog HDLでは8ビットになる

表2
配列記述の比較

	SystemVerilog	Verilog HDL
最大アクセス・ブロック	64ビット	8ビット
スライス・インデックス 例： `assign dataout [3] [1:0] = DATA [2] [3:2];` `//1度に4バイト(32ビット)をアクセス`	対応	非対応
ループで使用する初期データ・アレイ	対応	非対応
すべての256ビットのブロックを1度にアクセス可能に変更 例： `logic [3:0] [3:0] [1:0] [7:0] DATA`	対応	非対応
配列内のビットと部分の選択	対応	対応

リスト4　ステート・マシン記述

```
module FSM1(   input bit [7:0] in, input bit clk, reset, output bit [7:0] out_encoded, out_decoded);

bit [7:0] intermediate;
typedef enum bit[3:0] {s0 = 4'b0000, s1 = 4'b0001, s2 = 4'b0010, s3 = 4'b0011,
                       s4 = 4'b0100, s5 = 4'b0101, s6 = 4'b0110, s7 = 4'b0111,
                       s8 = 4'b1000, s9 = 4'b1001, s10 = 4'b1010, s11 = 4'b1011,
                       s12 = 4'b1100, s13 = 4'b1101, s14 = 4'b1110, s15 = 4'b1111} STATE;

STATE current_satate;

  encode a(in, current_satate, out_encoded);
  decode b(intermediate, current_satate, out_decoded);

assign intermediate = out_encoded;

always_ff @ (posedge clk)
begin
  if(reset)
    current_satate = current_satate.first();
  else
    current_satate = current_satate.next();
end
endmodule

module encode (input bit [7:0] in,
               input bit [3:0] current_satate,
               output bit [7:0] out_encoded);
bit       [7:0] lock;
assign out_encoded = in ^ lock;

always_comb
begin
  unique case (current_satate)
      0000: lock = {7{current_satate[0]}};
      0001: lock = {7{current_satate[0]}};
      0010: lock = {7{current_satate[0]}};
      0011: lock = {7{current_satate[0]}};
      0100: lock = {7{current_satate[1]}};
      0101: lock = {7{current_satate[1]}};
      0110: lock = {7{current_satate[1]}};
      0111: lock = {7{current_satate[1]}};
      1000: lock = {7{current_satate[2]}};
      1001: lock = {7{current_satate[2]}};
      1010: lock = {7{current_satate[2]}};
      1011: lock = {7{current_satate[2]}};
      1100: lock = {7{current_satate[3]}};
      1101: lock = {7{current_satate[3]}};
      1110: lock = {7{current_satate[3]}};
      1111: lock = {7{current_satate[3]}};
    default: lock = 8'b0;
  endcase
end
endmodule
```

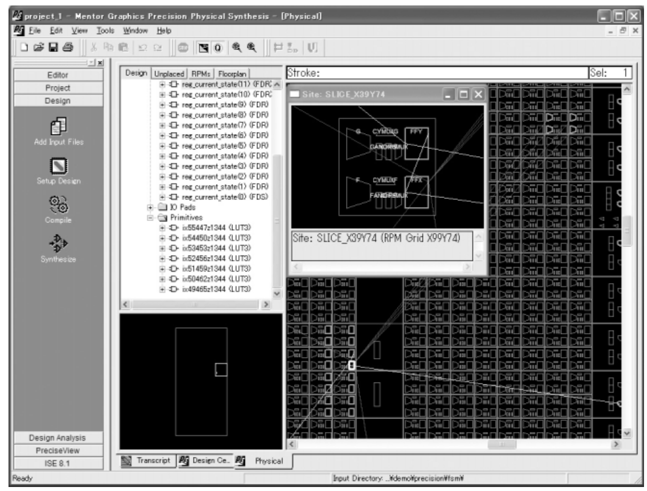


図5　ステート・マシン合成後のレイアウト情報の表示
FPGA内部のスライス（論理セル）の配置と，スライス間の結線情報を表示している

は，合成後に表示されるレイアウト情報の例です．

1）抽象データ・タイプ

SystemVerilogは，Verilog HDLと異なるハードウェア中心のネット，および4状態の論理値を持った変数データ・タイプをサポートしています．また，抽象度の高いモデルを表現するためのいくつかのデータ・タイプも追加されました．以下にその例を示します．

● struct

structを利用すると，複数の変数を共通の名まえでグループ化できます．これらの変数は，どの変数に対しても独立に割り当てることができます．また，すべてのグループを1回の宣言で割り当てられます．

● union

unionを利用すると，同じ変数ローケーションのさまざまな表現が可能となります．これにより，アクセスと一般的な設計のモデル化が容易になります．

● enum

enumを利用すると，ユーザ定義の列挙タイプを記述できます．これにより，データ・タイプ，またはデータ値なしの抽象的なタイプの変数を宣言します．後から設計への定義を必要とするデータ・タイプと値を追加できます．

```
enum {red, yellow, green} light1,
            light2; // anonymous int type
```

● キャスト代入

type，size，signedによるサブ表現によって，キャスト代入が可能となります．

```
type out = byte'(var1) + 8'b11111100;
size out = 3'(var2) -3'b111;
signed out = signed'(var1) -3'sb0011;
```

●ユーザ定義タイプ

SystemVerilogでは，ユーザが新しいデータ・タイプを定義できます．typedef（C言語とほぼ同じ）を使用して，新しいデータ・タイプを定義します．

```
typedef byte unsigned ubyte;
ubyte a, b;
```

2）変数に対するルールの緩和

SystemVerilogでは，変数の使用に関するルールが緩和されています．HDL設計者は，手続き文と連続代入の両方で変数を使用できます．すべての変数が一つのドライバを持つ場合，データ・タイプlogicを宣言します．この拡張により，「変数（Verilog HDLのreg）とネット（Verilog HDLのwire）のいずれを使用するべきか」といった混乱が解消されます．

変数については，以下のことが可能です．

- 手続き代入ステートメントの任意の数による値の割り当て
- 1回の連続代入ステートメントによる値の割り当て
- 一つのプリミティブの出力への接続
- 一つのモジュール・ポートの受信側への接続

こうしたルールの緩和により，モデルの作成が単純になります．変数が値を受け取る方法と関係なく，ほとんどすべての信号を変数として宣言できます．信号がマルチドライブ（例えば，双方向ポート）の場合はnetデータ・タイプが必要になります．

SystemVerilogのルールでは，変数の値のソースが単一であることが要求されます．例えば，変数が代入式の左辺で使用され，さらに誤ってモジュールの入力ポートに接続された場合，エラーが出力されます．Verilog HDLの場合も，このような状況（マルチドライブのコンパイル）でnetタイプが要求されますが，最終的には機能エラーとなります．

3）uniqueとpriority

SystemVerilogでは，分岐を判定するステートメントがユニーク（一意的）であるか，優先的な評価が必要かを，キーワードuniqueやpriorityを使用して指定できます（**リスト5**）．uniqueやpriorityといったキーワードは，シミュレータや論理合成ツール，そのほかのツールに対して，ユーザが意図するハードウェア・タイプを指示します．ツールは，コードが要求された論理を正しくモデル化しているかどうかのチェックにこの情報を使用します．

priority判定変更子が指定されているとき，ツールはソース・コードの判定の順序を維持しなければなりません．さらに，それらの判定が評価されて分岐が実行されない場合は，すべてのツールがエラーを出力しなければなりません．

unique判定変更子が指定されているとき，ツールは判定基準以外の最適化を行うことができます．しかし，二つのコード分岐が同時に真と評価された場合は，すべてのツールがエラーを出力しなければなりません．さらに，それらの判定が評価されて分岐が実行されない場合も，すべてのツールがエラーを出力しなければなりません．

リスト5　uniqueとpriority

```
priority if (a[2:1] = = 0) y = in1;
                                    // a equals to 0 or 1
  else if (a[2] = = 0) y = in2;    // a equals to 2 or 3
  else y = in3;           // a equals to any other value

unique case(a)
  0, 1: y = in1;
  2: y = in2;
  4: y = in3;
endcase
  // unspecified values will result in a compiler error
```

●論理合成技術の進化に設計言語も追随

SystemVerilogは，論理合成に関する，現在の効率の低いコーディング・スタイルを改善してくれます．これにより，コンパクトで簡潔なRTL記述を作成できるようになり，設計生産性が向上します．

論理合成技術が，ブール代数式のマッピングから始まり，やがて複雑なRAMやDSP，ステート・マシンといった機能のマッピングが可能となったように，設計言語も，今日の設計者が先進の機能を記述できるように進化しています．

Roger Do
米国Mentor Graphics社
FPGA Solutions Channel Marketing